

Project Implementation  
COS 301 Buzz Project  
Group: Testing Phase: Resources  
Version 1.0

Carla de Beer 95151835  
Prenolan Govender 13102380  
Shaun Meintjes 13310896  
Collins Mphahlele 12211070  
Dumisani Msiza 12225887  
Joseph Murray 12030733  
Sifiso Shabangu 12081622  
Joseph Potgieter 12003672  
Johan van Rooyen 11205131

*<https://github.com/Carla-de-Beer/Testing-Resources>*

University of Pretoria

24 April 2015

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Functional Testing</b>	<b>2</b>
2.1	Resources A . . . . .	2
2.1.1	Pre-condition violations . . . . .	2
2.1.2	Post-condition violations . . . . .	3
2.1.3	Data structure requirements . . . . .	4
2.2	Resources B . . . . .	5
2.2.1	Pre-condition violations . . . . .	6
2.2.2	Post-condition violations . . . . .	8
2.2.3	Data structure requirements . . . . .	8
<b>3</b>	<b>Non-functional testing / assessment</b>	<b>9</b>
3.1	Resources A . . . . .	9
3.1.1	Usability . . . . .	9
3.1.2	Performance . . . . .	9
3.1.3	Scalability . . . . .	10
3.1.4	Testability . . . . .	10
3.1.5	Security . . . . .	10
3.1.6	Reliability . . . . .	11
3.1.7	Reusability . . . . .	12
3.1.8	Pluggability . . . . .	12
3.2	Resources B . . . . .	12
3.2.1	Usability . . . . .	13
3.2.2	Performance . . . . .	13
3.2.3	Scalability . . . . .	13
3.2.4	Testability . . . . .	13
3.2.5	Security . . . . .	13
3.2.6	Reliability . . . . .	13
3.2.7	Reusability . . . . .	13
3.2.8	Pluggability . . . . .	13
<b>4</b>	<b>Critical Evaluation and Recommendations</b>	<b>14</b>
4.1	Resources A . . . . .	14
4.1.1	Master specification compliance . . . . .	14
4.2	Resources B . . . . .	14
4.2.1	Code and code structure legibility . . . . .	14
<b>5</b>	<b>References</b>	<b>14</b>

# 1 Introduction

The purpose of this task was to test the functionality provided by the Resources teams for the Buzz Project.

According to the master specification document, version 0.1, released 13 March 2015 (SOLMS *et al*, 2015), the buzzResources module is used to upload and manage resources like media (e.g. images, video) and documents (e.g. PDF documents, Open Document Format documents). These resources are to be either embedded or linked to in posts.

This team took each of the pre and post-conditions of the required use case, considering the work of both Resources A and Resources B, and tested them for compliance with the functional requirements.

The use case for this team is called **uploadResource**. In terms of the requirements for this use case, and as defined by the master specification document, users should be able to upload resources such as media files or documents. Any uploaded resource should be accessible by other users who should also be able to specify links to that resource.

The functional requirements preconditions for this use case included the need to

- detect the mime type
- check that size constraints are met
- check that the resource type is supported

The functional requirements postconditions for this use case included the need to

- check that the resource persisted
- check that the URL for the resource was created

**Note to the lecturers: the testing resources group had to import the Resources A code repository into the GitHub repository to allow access to the code to all members. This may lead to the appearance of a skew in the git contributions, when this is not the case.**

## 2 Functional Testing

### 2.1 Resources A

#### 2.1.1 Pre-condition violations

##### **uploadResources**

- The mime-type can be detected before the file is uploaded, thus not in violation of condition.

- The size can be determined before the file is uploaded, also not in violation of pre-condition.
- The database is queried (albeit erroneously due to logic faults) and checked if the mime-type is supported before the file gets uploaded.

**getResourcesBySpaceId** The pre-condition will be that the SpaceID that is sent in as a parameter does exist in the database prior to this execution. There are no violations of this pre-condition.

**getResourcesAll** The pre-condition will be that there exist resources in the database prior to this execution. There are no violations of this pre-condition.

**getResourcesRelated** The pre-condition will be that the appraisal type (RelatedID) that is sent in as a parameter does exist in the database prior to this execution. There are no violations of this pre-condition.

**removeResource** The pre-conditions which in this case is resource ID being passed as a parameter to removeResource, the ID must exist in the database for a resource to be removed.

**addResourceType** The pre-condition will be the MIME type passed would be a valid and existing MIME type and the size limit would be a valid size which would be stated in bytes.

The pre-condition can be violated by passing the MIME type that does not exist and the function will just add the size limit and the MIME type to the Resources constraints in the database.

**removeResourceType** The objectID passed as a parameter must conform to a hexadecimal sequence. If the objectID specified is not hexadecimal, the function returns false.

**updateResourceType** The pre-conditions which in this case is the constraintID passed as a parameter which is the ID of the constraint to be updated must exist in Resources constraints in the database and sizeLimit which is the new size limit of the constraint must be stated in bytes.

### 2.1.2 Post-condition violations

#### uploadResources

- The wrong type of file and file size is added to the database due to improper testing of the mime-type and size.
- A URL to the resources are not explicitly created, the database needs to be queried each time a resource needs to be found.
- Persisting resource condition not violated, the resources persist.

**getResourcesBySpaceId** The post-condition will be that the resources that exist in a certain space will be returned. This post-condition fails, because the function queries the database and returns all of the resources in the database, which is incorrect. No exceptions are thrown if the Buzz Space does not exist.

**getResourcesAll** The post-condition will be that all of the resources in the database will be returned. This post-condition holds, and all of the resources are returned.

**getResourcesRelated** The post-condition will be that the resources that exist in the database of the specified appraisal type will be returned. This post-condition fails, because the function queries the database and returns all of the resources in the database, which is incorrect.

**removeResource** The post condition will be the removal of the specified resource by ID in the database.

The removeResource does not conform to the post-condition as it cannot remove the resource that exist in the database with the specified ID.

**addResourceType (addConstraint)** The post-condition is when the size limit and the MIME type would be added to the Resources constraints in the database.

**removeResourceType (removeConstraint)** The objectID which is the ID of the constraint to be removed from Resources constraints in the database will be removed.

**updateResourceType (updateConstraint)** The constraint with the specified constraintID in Resources constraints in the database will be updated by the new size limit.

### 2.1.3 Data structure requirements

**uploadResources** No data structure requirements were given to this function apart from the fact that a timestamp had to be appended to the file name in order to prevent file name clashes. It excels in this area, providing a correct and meaningful timestamp and appending it to the back of the file name.

**getResourcesBySpaceId** No data structure requirements mentioned in the service contract, nor does getResourcesBySpaceId utilise any data structures.

**getResourcesAll** No data structure requirements mentioned in the service contract, nor does getResourcesAll utilise any data structures..

**getResourcesRelated** No data structure requirements mentioned in the service contract, nor does getResourcesRelated utilise any data structures.

## 2.2 Resources B

None of the functions or use cases for Resources B make use of callbacks.

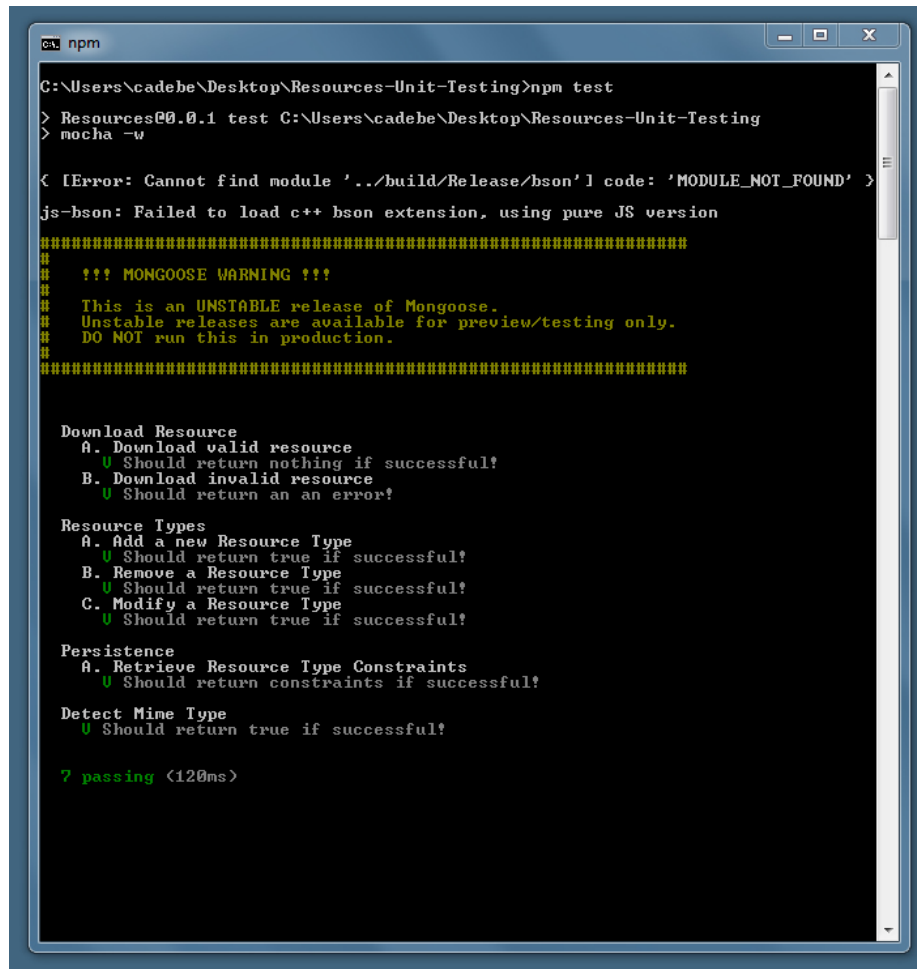
*"Nearly everything in node uses callbacks [...] Callbacks are functions that are executed asynchronously, or at a later time. Instead of the code reading top to bottom procedurally, async programs may execute different functions at different times based on the order and speed that earlier functions like http requests or file system reads happen."*

[github.com/maxogden/art-of-node#callbacks](https://github.com/maxogden/art-of-node#callbacks)

The implementation code makes an attempt to allow for the creation of modules for export, but since all the functions, including the main use case, are all object or value-returning functions, it can be said that the code fails, outright, all its services contracts. This is due to the fact that the Resources B code, as it currently stands, cannot be unit tested or integrated into the main code framework for the project. The Testing Resources B team undertook an attempt to write unit testing code, working with the existing code structure, rather than amending it other than by introducing a callback to the `uploadResources` function, in order to be able to test the code for compliance with both functional and non-functional requirements. The code did not run and terminated in an error message stating "Error: Cannot find module 'DataBaseStuff'". It seems, therefore that one of the modules is missing from the code structure and, as such, the code could not be tested by means of callbacks. After some rummaging around in the various GitHub branches, squirrelled away in one of the team members' branches was a section called "Unit Testing Code". It became clear that this code is intended to be run via node package manager (npm) and ran unit tests for the following options:

- Download valid resource
- Download invalid resource
- Add a new Resource Type
- Remove a Resource Type
- Modify a Resource Type
- Retrieve Resource Type Constraints
- Detect Mime Type

In the absence of being able to test via modules and callbacks, the testing team used this unit test code, as provided by the implementation team, for the testing the Resources B section. Below is a screenshot of the result obtained from the initial running of the provided unit testing code, prior to changing input parameters.



```
ca npm
C:\Users\cadebe\Desktop\Resources-Unit-Testing>npm test
> Resources@0.0.1 test C:\Users\cadebe\Desktop\Resources-Unit-Testing
> mocha -w

< [Error: Cannot find module '../build/Release/bson' code: 'MODULE_NOT_FOUND' ] >
js-bson: Failed to load c++ bson extension, using pure JS version
#####
!!! MONGOOSE WARNING !!!
This is an UNSTABLE release of Mongoose.
Unstable releases are available for preview/testing only.
DO NOT run this in production.
#####

Download Resource
  A. Download valid resource
    ✓ Should return nothing if successful!
  B. Download invalid resource
    ✓ Should return an an error!

Resource Types
  A. Add a new Resource Type
    ✓ Should return true if successful!
  B. Remove a Resource Type
    ✓ Should return true if successful!
  C. Modify a Resource Type
    ✓ Should return true if successful!

Persistence
  A. Retrieve Resource Type Constraints
    ✓ Should return constraints if successful!

Detect Mime Type
  ✓ Should return true if successful!

7 passing (120ms)
```

Figure 1: Unit testing Resources B: the output provided by running the unit test code, as provided by the implementation team

## 2.2.1 Pre-condition violations

**uploadResources** The uploadResources use case function makes an attempt to fulfil the mimetype detection precondition, as well as the getResourceTypeConstraints and throws an exception in the case where

- the file size constraints have not been met
- the resource type is not supported
- the mimetype could not be detected

The raising of these exceptions is commensurate with the services contract requirements, as specified in Figure 28 in the master specification document (SOLMS et al, 2015). There is no actual unit testing code written for the uploadResources use case. Considering that the other functions pertaining to the uploading of

resources do not work, it is expected that this function will fail. In terms of the actual code, the mimetype size is tested by means of the line:

```
file.size <= (constraints.maximumSize * 1000)
```

There is no explanation as to why this particular value has been chosen as the file size constraints. Something similar to the notion of a constant value, declared at the head to the file, might have been more appropriate and explanatory in this case. In summary, there is an attempt to fulfil the use case pre-conditions by means of inclusion of exceptions for the cases where the preconditions may fail, however, the code fails in cases where erroneous input is added. In such cases the code would have benefited from the application of additional exception throwing to cover these scenarios.

**downloadResource** This use case calls the line

```
var results = resources.downloadResource("mountain.jpg");
```

and when this line is changed to read

```
var results = resources.downloadResource("mountain");
```

the code still executes successfully. It appears that the unit testing for this function is not working correctly, and as such, the use case cannot be tested with the code provided.

**modifyResourceType** The required pre-conditions for this use case include the need for testing the validity of the mimetype that is sent through, as well as a parameter to limit the size limit the size of the mimetype change, presumably to avoid potential malpractice. The unit testing code as provided by the implementation team was not considered to be reliable in the testing of this use case, and since the modifyResourceType function appears to be a Boolean returning function, rather than actually modifying the specified item in the database, it can be concluded that this use case is violating its required pre-conditions.

**addResourceType** The pre-condition of addResourceType is assumed to be valid data. Tests for empty strings and resource types that do not belong to the MIME types were passed, without exceptions being thrown.

**removeResourceType** The pre-condition of removeResourceType is assumed to be valid data. Tests for empty strings and resource types that do not belong to the MIME types were passed, without exceptions being thrown. There were also no exceptions thrown for types that have not been added.

**removeResource** The pre-condition of removeResource is assumed to refer to the identification number of a resource in the database which should exist prior to this execution. There are no violations of this pre-condition due to error checks being performed to verify whether or not the entry exists.

**removeResourceTypeConstraint** The pre-condition of removeResourceType-Constraint is assumed to be valid data. Tests for empty strings were passed without exceptions being thrown. There were also no exceptions thrown for type constraints that have not been added.



**detectMimeType** There appears to be problems with the mimetype detection functionality. It is hard to see how this function, as it currently stands, will return a value other than true. Running the provided test code for the “Detect Mime Type” seems to confirm this. Each time the code was run, the code executes without fail when tested against various mimetypes. Representatives from the various mimetype groups were used, with emphasis on the ones that are expected to be most frequently used, including:

- image/jpeg
- image/png
- image/gif
- image/targa
- image/svg+xml
- image/example
- audio/mp3
- message/http
- multipart/alternative
- multipart/encrypted
- text/cmd
- text/csv

However, when the line `var results = mime.detectMimeType("./r/mountain.jpeg");` was amended to `var results = mime.detectMimeType("./r/rubbish");` the code also happily returned a true value. It seems that neither the unit testing code for this use case, nor the actual use case code, works correctly in that the mimetypes are not correctly detected.

### 2.2.2 Post-condition violations

**uploadResources** There is a folder with images and music files, but the code seems to make the presumption that there is a server in place that can handle the uploads. Since the `uploadResources` use case was not included in the unit testing, since the unit testing code has proved itself to be defective, and since many of the functionalities, such as mimetype detection, which `uploadResources` depends on, do not work themselves, it is concluded that the uploading of resources has not persisted to the database. No URLs seems to be created for the uploaded files.

**modifyResourceType** It is not clear whether the items uploaded to the database are actually persisting. There is a folder with images and music files, but the code seems to make the presumption that there is a server in place that can handle the uploads.

**removeResource** The post-condition of `removeResource` would be the removal of an entry. Due to a lack of unit testing functionality, the post-condition fails as there is no visible indication of `removeResource` working.

**downloadResource** Since resources could not be uploaded to the database, resources could consequently not be downloaded.

**detectMimeType** The mimetypes were not correctly identified.

### 2.2.3 Data structure requirements

**uploadResources** The `uploadResource` use case calls a filepath to an object to be uploaded, rather than an object or data structure itself. No data structures have been specified as required in the master specification document, and none were required.

**addResourceType** No data structure requirements mentioned in the service contract. However, `addResourceType` requires a string value as the only input. Said string must be a MIME type.

**removeResourceType** No data structure requirements mentioned in the service contract. However, `removeResourceType` requires a string value as the only input. Said string must be a MIME type.

**modifyResourceType** There are no data structure requirements for this use case.

**removeResource** No data structure requirements mentioned in the service contract, nor does `removeResource` utilise any data structures.

**downloadResource** The `downloadResource` use case calls a filepath to an object to be uploaded, rather than an object or data structure itself. No data structures were required for this use case.

**detectMimeType** The `detectMimeType` use case calls a filepath to an object to be uploaded, rather than an object or data structure itself. No data structures were required for this use case.

### 2.2.4 Pre-condition violations

**addResourceType** The `addResourceType` use case was specified as part of the use case diagram (p 28 of the master specification document (SOLMS, 2015)), however its services requirements were not explicitly stated in the document. It is assumed, though, that the implementation of this use case attempts to fulfil the pre-conditions which are the resource type, or mimetype, and the file size, and should throw an exception in the case where

- The mimetype is syntactically incorrect
- The file size is not a string containing integer values

Using the test code provided by the implementation team, the `addResourceType` test code runs without fail when tested against the various valid mime-Types and file sizes that include some of the following:

- `image/jpeg 3000`
- `image/png 4000`
- `image/gif 5000`
- `image/targa 200`
- `text/csv 100`

however when the line `it('Should return true if successful!' and results.should.equal(true);` was amended to `it('Should return false if successful!' results.should.equal(false);` and was tested against various invalid mimetypes and Files sizes such as the following:

- `jpeg/image 3000s`
- `png/image 4000s`
- `gid/image 5000s`
- `targa/image 200s`
- `csv/text 100s`

The test code failed as seen in the figure bellow

#### 2.2.5 Post-condition violations

**addResourceType** The mimetypes are added as a constraint, as required, but it is not clear to which field in the database, or file structure, the mimetypes and there sizes are added.

#### 2.2.6 Data structure requirements

**addResourceType** The function allows two strings as input parameters, and therefore there is no specific data structure requirement.

## 3 Non-functional testing / assessment

### 3.1 Resources A

The use case was tested against the following list of architectural requirements:

### 3.1.1 Usability

**uploadResources** Usage of this function is fairly easy and obvious. The selected files from the file selector only need to be sent through to the function and it handles all the rest of the heavy lifting. It can handle multiple files at once and tests each file individually.

**getResourcesBySpaceId** This function is easy and simple to use. The function is called with the Space ID, and a list of the resources for that space is returned.

**getResourcesAll** This function is simple and easy to use. The function is called and all resources within the database is returned.

**getResourcesRelated** This function is simple and easy to use. The function is called and an appraisal type is sent as parameter. All resources of that appraisal type is returned.

### 3.1.2 Performance

**uploadResources** The performance of this function would rely on network speed and stable connection to the database, since it does not require much computational time or pc resource allocation.

**getResourcesBySpaceId** This function does not suffer from performance issues. The performance of this function depends on the connection and communication speed of the database.

**getResourcesAll** This function does not suffer from performance issues. The performance of this function depends on the connection and communication speed of the database.

**addingConstraint,removeConstraint and updateConstraint** Connection to the database takes longer adding Constraint response time is unrealistic for the system as it takes longer.

**getResourcesRelated** This function does not suffer from performance issues. The performance of this function depends on the connection and communication speed of the database.

### 3.1.3 Scalability

**uploadResources** This function allows multiple files to be uploaded together, with no specified limit of files. The size of each file is limited by the relevant mime-type entry in the database, thus, theoretically, any file size should be able to be uploaded, if the database permits it. Furthermore, multiple users can use the function concurrently, because of the way the server can be set up.

**getResourcesBySpaceId** This function does not place any restriction on the possible size of the database, as a list of all resources within a certain space is returned.

**getResourcesAll** This function does not place any restriction on the possible size of the database, as a list of all resources in the database is returned.

**getResourcesRelated** This function does not place any restriction on the possible size of the database, as a list of resources of a certain type is returned.

#### 3.1.4 Testability

**uploadResources** The testability of this function is not hard at all, since it allows only a certain number of things to happen. If a file is not allowed, it shouldn't be uploaded, if it is, it should. If the file size is too large, likewise. It is quite easy to send a query to the database to see if the correct information was stored by the function, thereby further enhancing the testability thereof.

**getResourcesBySpaceId** This function is easy to test, as it can be tested from the interfaces that was created.

**getResourcesAll** This function is easy to test, as it can be tested from the interfaces that was created.

**getResourcesRelated** This function is easy to test, as it can be tested from the interfaces that was created.

**addingConstraint,removeConstraint and updateConstraint** No unit test created for testing of each functions implemented and provided testing is in the app.js file which caters the interface side of the function,however adding Constraint and other functions result in too long to response exception being thrown by the browser as connection with the database is not well established and fails.

#### 3.1.5 Security

**uploadResources** This function does not handle any type of security (except not allowing harmful mime-types). The connection to the database, for instance, is created somewhere else and the database object is only used by this function.

**getResourcesBySpaceId** This function is secure, as it only retrieves information from the database, about the resources for a certain space.

**getResourcesAll** This function is secure, as it only retrieves information from the database, about all of the resources in the database.

**getResourcesRelated** This function is secure, as it only retrieves information from the database, about all of the resources of a certain appraisal type.

**addingConstraint** Interface is not secured in the client side adding Constraint section does not validate user input and adds the given input to the database without any validation ,the only validation is on the server side where it checks if the input is not already in the database.

### 3.1.6 Reliability

**uploadResources** In the cases tested, the function is not reliable as it should be. It allows multiple files to be uploaded, as it should, however - it allows ALL files of any size to be uploaded due to a logic error in the code. The size is tested the wrong way around (a test is done to see if the max size is smaller than the file size, which is wrong). Not only that, a function is used to count the amount of records found, but the object containing all the records found cannot be "counted" in the way the coder wanted to. In order to make sure that the code was indeed faulty, new code was written to test the file to be uploaded and it was found that the original coder had indeed made a mistake.

**getResourcesBySpaceId** This function is very unreliable, because it does not conform to the post-condition. It returns a list of all of the resources, instead of only returning a list of resources within a certain space.

**getResourcesAll** This function is reliable as it conforms to the post-condition. The function returns a list of all of the resources within the database.

**getResourcesRelated** This function is very unreliable, because it does not conform to the post-condition. It returns a list of all of the resources, instead of only returning a list of resources of a certain appraisal type.

**addingConstraint,removeConstraint and updateConstraint** Lack of input validation results in garbage data entered in the database,which result in unreliable results and connection to database takes too long or results in browser throwing an exception for taking too long to respond,thus module is not reliable to complete a task base on this factors.

### 3.1.7 Reusability

**uploadResources** This function can easily be reused, since it keeps no remnants of previous data that can possibly corrupt new data. Each use of the function is thus a clean slate, so to speak. It is also exported as part of the package.

**getResourcesBySpaceId** This function is reusable, as it retrieves information from the database, about the resources for a certain space. This function can be called and reused throughout the system.

**getResourcesAll** This function is reusable, as it only retrieves information from the database, about all of the resources in the database. This function can be called and reused throughout the system.

**getResourcesRelated** This function is reusable, as it only retrieves information from the database, about all of the resources of a certain appraisal type. This function can be called and reused throughout the system.

### 3.1.8 Pluggability

**uploadResources** This function is pluggable to the extent that mongoose is used as an interface to connect to the database. When that requirement is complied with, the function will be portable (it will accept file data and write it to the connected database). Electrolyte is also used to make the function more pluggable.

**getResourcesBySpaceId** This function is pluggable, as it retrieves information from the database, about the resources for a certain space. This function can be called and reused within any system if called on a database that has the same structure as the current database.

**getResourcesAll** This function is reusable, as it only retrieves information from the database, about all of the resources in the database. This function can be called and reused within any system if called on a database that has the same structure as the current database.

**getResourcesRelated** This function is reusable, as it only retrieves information from the database, about all of the resources of a certain appraisal type. This function can be called and reused within any system if called on a database that has the same structure as the current database.

## 3.2 Resources B

The use case was tested against the following list of architectural requirements:

### 3.2.1 Usability

**modifyResourceType** Since the function does not work as required, any associated performance measurement, in this case, is irrelevant.

**removeResource** removeResource demonstrates usability by being simple and easy to use. The function only needs be called along with the ID of the resource to be removed which is intuitive.

**addResourceType** There is no indication of the interface design for this use case, however the intension seems to be for the function to be fairly simple to use: only a mimetype and a maximum size is required.

addResourceType demonstrates usability by being simple and easy to use. The function only needs be called along with the string name of the resource to be added which is intuitive.

**removeResourceType** removeResourceType demonstrates usability by being simple and easy to use. The function only needs be called along with the string name of the resource to be removed which is intuitive.

**retrieveResourceTypeConstraint** retrieveResourceTypeConstraint demonstrates usability by being simple and easy to use. The function only needs be called along with the string name of the resource of which the constraint should be retrieved which is intuitive.

### 3.2.2 Performance

**removeResource** removeResource does not suffer from any performance issues in and of itself, however should the database maintain a sizeable stature, removeResource will be seen to suffer in performance. This is to be expected. performance

**addResourceType** Since the code could not be tested and correctly excuted, its performance could not be quantifiably ascertained.

addResourceType does not suffer from any performance issues in and of itself, however should the database maintain a sizeable stature, addResourceType will be seen to suffer in performance. This is to be expected. performance

**removeResourceType** removeResourceType does not suffer from any performance issues in and of itself, however should the database maintain a sizeable stature, removeResourceType will be seen to suffer in performance. This is to be expected. performance

**retrieveResourceTypeConstraint** retrieveResourceTypeConstraint does not suffer from any performance issues in and of itself, however should the database maintain a sizeable stature, retrieveResourceTypeConstraint will be seen to suffer in performance. This is to be expected. performance

### 3.2.3 Scalability

**modifyResourceType** A size limit is imposed on the degree of modification that can be made to the uploaded resource, as such there is a limitation on the scalability of modifications that can be made.

**removeResource** removeResource offers no solutions to accomodate an increase or decrease in size of the database.



**addResourceType** There is potential for scalability in this use case since there is no fixed library of mimetypes or restriction on the mimetypes that can be uploaded.

addResourceType offers no solutions to addResourceType an increase or decrease in size of the database.

**removeResourceType** removeResourceType offers no solutions to removeResourceType an increase or decrease in size of the database.

**retrieveResourceTypeConstraint** retrieveResourceTypeConstraint offers no solutions to retrieveResourceTypeConstraint an increase or decrease in size of the database. >>>>>> master

### 3.2.4 Testability

**modifyResourceType** The function has been modularised and this should , in principle, facilitate its testability.

**removeResource** removeResource has a complete lack of testability due to it not conforming to proper unit testing standards (making use of callbacks, depending on a live database etc.).

**addResourceType** The function was designed to be modularised, so as such this should improve testability, however, the unit test code always returns true, regardless of the mimetype input provided. The conclusion is that the unit testing code cannot be relied upon to provide accurate testing feedback in this case.

addResourceType seems to be testable.

**removeResourceType** removeResourceType seems to be testable.

**retrieveResourceTypeConstraint** retrieveResourceTypeConstraint seems to be testable.

### 3.2.5 Security

**modifyResourceType** This function may pose a security risk. It does not check the mimetype to see if it may be harmful. The function also doesn't place constraints on the new size limit, which could lead to a significant portion of the database's memory being allocated to the resources.

**removeResource** The only security concern would be an incorrect ID being used in order to remove a different resource than required, potentially putting the system at risk.

**addResourceType** This use case does not check for valid mimetypes. It accepts any text which could open a path SQL injection if the values are stored in a database.

addResourceType has no input validations and is therefore a great security risk.

**removeResourceType** removeResourceType has no input validations and is therefore a great security risk.

**retrieveResourceTypeConstraint** retrieveResourceTypeConstraint has no input validations and is therefore a great security risk.

### 3.2.6 Reliability

**modifyResourceType** The use case does not work as expected and cannot be described as reliable.

**removeResource** According to the failure of the post-condition, removeResource gives a poor indication with regards to reliability.

**addResourceType** Since the unit test code for this use case does not work correctly, the addResourceType functionality cannot be properly tested. The conclusion is that this use case may not be reliable.

According to the failure of the post-condition, addResourceType gives a poor indication with regards to reliability.

**removeResourceType** According to the failure of the post-condition, removeResourceType gives a poor indication with regards to reliability.

**retrieveResourceTypeConstraint** According to the failure of the post-condition, retrieveResourceTypeConstraint gives a poor indication with regards to reliability.

### 3.2.7 Reusability

**modifyResourceType** The use case does not work as expected and cannot be described as reusable.

**removeResource** removeResource is not very reusable due to the dependence on a single database. Using this function throughout this system would not be possible/practical.

**addResourceType** Due to the absence of use of callbacks in the code, as it currently stands, the code cannot be integrated into the existing implementation structure, and is therefore not reusable.

addResourceType is not very reusable due to the dependence on a single database. Using this function throughout this system would not be possible/practical.

**removeResourceType** removeResourceType is not very reusable due to the dependence on a single database. Using this function throughout this system would not be possible/practical.

**retrieveResourceTypeConstraint** retrieveResourceTypeConstraint is not very reusable due to the dependence on a single database. Using this function throughout this system would not be possible/practical.

### 3.2.8 Pluggability

**modifyResourceType** Due to the absence of callbacks in this code, as it currently stands, the code cannot be integrated into the existing implementation structure and is therefore, cannot be considered to be pluggable.

**removeResource** In the same vein as reusability, removeResource cannot be integrated into other systems due to a lack of robustness and versatility.

**addResourceType** Due to the absence of use of callbacks in the code, as it currently stands, the code cannot be integrated into the existing implementation structure, and is therefore not pluggable.

In the same vein as reusability, addResourceType cannot be integrated into other systems due to a lack of robustness and versatility.

**removeResourceType** In the same vein as reusability, removeResourceType cannot be integrated into other systems due to a lack of robustness and versatility.

**retrieveResourceTypeConstraint** In the same vein as reusability, retrieveResourceTypeConstraint cannot be integrated into other systems due to a lack of robustness and versatility.

## 4 Critical Evaluation and Recommendations

### 4.1 Resources A

#### 4.1.1 Master specification compliance

The following of the master specification for the implementation was followed correctly to some extent in the resource implementation ,most functions were

implemented and worked to some extent, functionality was prioritised over following software principles like having a unit test for each function created and being specific in documentation of functions parameter types .

## **4.2 Resources B**

### **4.2.1 Code and code structure legibility**

The code has some degree of JSDoc comments, but could have benefited from a more comprehensive commenting system so as to allow newcomers to the code to better understand the program logic. The use of an explanatory README file would also have been welcomed by the testing team and others new to the work to help understand the logic behind the system implemented. Thus, from a code legibility point of view, there are some improvements that can be made to make the existing code more user-friendly. On a positive note, the code does make use of modularisation in the sense the additional functionalities are separated out from the main use case, making it easier to understand, and perhaps also easier to test the code. In terms of the files themselves, the file structure could have benefited from a better organisational structure. There are files with duplicate naming, for example, residing in different folders, making it unclear how these fit into the existing code structure (it was assumed that the code folders in the master branch were the most current). In addition, the unit testing code was hidden away in a branch and, as such, was not that straightforward to find.

## **5 References**

SOLMS, F., PIETERSE V., OMELEZE S., RAMASILA, L. 2015. *Buzz Discussion Board Requirements and Design Specifications (version 0.1)*. Department of Computer Science, University of Pretoria.

Figure 2: Unit testing Resources B: the output provided by running the unit test code, as provided by the implementation team with invalid inputs